文章编号: 1000-5641(2023)05-0065-12

FeaDB: 基于内存的多版本在线特征存储

高 歌, 胡卉芪

(华东师范大学 数据科学与工程学院, 上海 200062)

摘要: 特征管理是搭建人工智能数据管道中的重要一环. 特征存储要求在模型训练和推理阶段提供有效版本的特征推送服务. 为响应这一需求, 特征存储需要为特征实时更新和版本管理提供保证, 以协同上游的特征摄取, 为模型服务系统提供数据动力. 在人工智能辅助决策的在线预测任务中, 为了提供更好的用户体验, 模型服务系统需要实时响应决策请求, 实时特征检索面临更低延迟的挑战. 聚焦这一挑战, 开发基于内存的多版本在线特征存储 FeaDB. 使用时间序列建模特征, 并提供特征版本管理语义, 满足特征从生产到消费的版本管理需求; 采用追加写方式保证实时特征加载性能, 设计基于版本的索引减少读延迟; 为进一步减小特征消费延迟, 提出版本快照机制, 实验证明采用快照读机制增加了特征集版本的检索效率.

关键词: 人工智能数据管理系统; 多版本存储; 在线特征存储

中图分类号: TP392 **文献标志码**: A **DOI**: 10.3969/j.issn.1000-5641.2023.05.006

FeaDB: In-memory based multi-version online feature store

GAO Ge, HU Huiqi

(*School of Data Science and Engineering, East China Normal University, Shanghai* 200062, *China*)

Abstract: Feature management plays an important role in the AI(artificial intelligence) pipeline. Feature stores are designed to offer effective versioning of features during the model training and inference stages. Feature stores must ensure real-time feature updates and version management to collaborate with the upstream data ingestion tasks and power the model serving system. In AI-powered online decision augmentation applications, the model serving system responds to requests in real time to provide better user experience, and feature stores face the challenge of low-latency online feature retrieval. Focusing on this challenge, we developed FeaDB, an in-memory based multi-version online feature store, which adopts a time series model and provides feature versioning semantics to automatically manage features from ingestion to serving. Moreover, an append-write operation was applied to ensure ingestion performance, and version indexing was optimized to improve read operations. A snapshot mechanism is proposed, and it was experimentally proven that snapshot read operations improve performance of lookup and range lookup.

Keywords: database for Artificial Intelligence; multi-version store; online feature store

0 引 言

人工智能 (artificial intelligence, AI) 应用普遍存在于日常生活中. 然而, 应用的开发、部署、维护以及 AI 数据管理的复杂度为构建现实世界的 AI 应用带来了极大的成本. 通常, 企业为统一化开发流

收稿日期: 2023-06-30

基金项目: 上海市自然科学基金 (23ZR1418300)

通信作者: 胡卉芪, 男, 副教授, 研究方向为数据库系统、分布式系统. E-mail: hqhu@dase.ecnu.edu.cn

程, 将机器学习任务划分为一系列有序阶段, 主要涵盖数据收集、数据准备、特征工程、模型训练、模型验证、部署和监控等关键步骤, 并将配置和工具等基础设施集成到系统中, 以自动化线下测试与线上生产环境的迭代并改善协作; 由此构建的一套机器学习工作流称为机器学习管道 (machine learning pipeline)[1-4]. 在管道构建和维护阶段, 模型的开发和部署只占小部分, 隐藏很多针对机器学习数据管理的“技术债务”[5], 需要耗费大量的数据管理成本; 同时, 随着管道规模增加, 不同任务间特征数据的组合、演化过程愈加复杂, 管道中缺乏对不断更新的特征版本的一致性管理. 通过将中心化特征平台集成于数据管道中: 特征转换引擎处理特征工程任务, 特征存储缓存预计算的特征来追踪特征演化过程; 并向下游模型提供特征消费 (feature serving, 与后文的“推送”“检索”“查询”为同义) 服务. 实现特征从发布、生产到推送的一致性管理, 改善了机器学习生产流程, 提升机器学习生态系统的扩展型和鲁棒性.

现今开发的特征管理平台是批处理、流计算、缓存和存储的集成系统. 工业界通常使用开源存储在机器学习管道之上构建特征存储: Uber[6] 为规范化可扩展模型训练到生产服务容器部署的统一管理, 采用集成的开源系统和内部开发组件构建了机器学习平台, 将其用于 UberEats 等 AI 应用中; Feast[7] 采用典型的批流架构的离/在线存储协同进行特征管理的平台, 提供注册、存储、消费、监控的能力; Feast 不直接管理存储, 通过增量物化过程将数据从离线存储加载到在线存储, 保证训练和推理阶段的特征的一致性. 学术界从特征转换、可扩展性管理、新鲜度与成本之间的权衡、异构特征管理等不同角度开展了研究: 特征提取是许多在线决策增强 (on-line decision augmentation, OLDA) 任务中最耗时的工作, 开发基于持久性内存的分布式内存数据库系统[8], 设计经过 PMEM (persistent memory) 优化的持久性双层跳表索引, 能有效减少时间窗口特征抽取时间. 当特征在训练和推理阶段有不同实现方式时, 特征重用引入了额外的同步开销, 为临时构建的机器学习管道增加了复杂度, Ormenisan 等[9] 讨论关于管道可扩展性的问题, 并构建了具有水平可扩展的特征存储作为机器学习管道的一部分. 除了表格型数据的特征, 在自监督深度学习领域中嵌入向量成为需要管理的对象, Embedded Store[10] 总结了管理嵌入训练数据, 衡量嵌入特征质量和监控使用嵌入的下游模型带来的挑战, 讨论了以嵌入向量作为管理对象的特征管道构建方法.

特征存储除了要满足上述管理诉求以外, 特征新鲜度也成为影响模型性能的主要因素之一. 特征新鲜度, 即版本更新在时间维度上的新旧程度. 特征存储需要推送满足新鲜度约束的特征; 尤其在基于在线决策或事件驱动的在线预测等实时机器学习任务中, 模型推理阶段需要在线特征存储推送实时数据来生成更准确的预测, 并使模型适应不断变化的环境. 例如, 实时个性化推荐服务[11]: 根据用户的实时行为和反馈, 实时分析用户兴趣和偏好, 并向其推荐相关商品、新闻等内容. 信用欺诈检测任务: 训练实时机器学习模型来分析用户的历史交易和实时交易行为是否存在异常变化, 以快速识别潜在欺诈交易[8]. 送餐时间估计: 结合历史、静态特征 (如商家在该区域送餐的平均时间) 和基于环境变化的实时数据 (路线距离、交通拥堵情况和实时行进速率) 即时反馈到达时间[12]. 综合上述应用特点, 实时预测任务处理的主要特征有: ① 预测是基于事件驱动的, 请求触发预测同步处理, 并将结果并反馈给用户. ② 模型推理结果需要满足低延迟和时效性以确保模型在最新的数据集中的处理决策. 而同步特征计算带来的延迟是不可忍受的, 通常采用异步计算——频繁更新的特征流预缓存到在线特征存储中, 事件到来时查询最新相关特征到下游任务. ③ 一批特征推送可能同时包含实时特征和静态 (与时间无关) 或历史特征 (查询非最新特征), 特征查询要保证查询延迟与当前版本数量及查询新鲜度无关. ④ 特征版本消费频率随时间推进减少, 且在线特征访问具有长尾分布的特点. 大量旧版本和非热点数据作为冷数据需要被回收. 基于上述特性, 本文考虑了在线特征存储系统中的主要设计要点: ① 多版本特征管理, 支持特征发布和版本控制, 并提供特征版本查询. ② 支持大规模特征的高吞吐批

量更新, 实现特征消费的低延迟和高时效性. ③ 查询延迟和新鲜度无关, 加速最新特征读, 同时不影响历史版本检索效率. ④ 根据特征访问的滞后淘汰性和长尾分布特点, 设计冷数据回收机制.

工业界对于在线特征存储的解决方案通常基于开源存储系统, 如内存存储 Redis、键值存储 CockroachDB 等. 这些开源方案存在一定局限性: 缺乏对版本控制功能的原生支持; 虽然内存存储如 Redis 等可以提供十万级 QPS (queries per second) 的读写吞吐, 但查询效率取决于系统实现, 新鲜度约束的查询可能影响历史特征的查询效率, 并难以进行针对性优化; 不支持冷热数据区分且大量旧版本需要采用手动回收策略. 典型的在线存储优化方案有: Doordash 基于 Redis 设计特征存储并应用于送餐时间预测任务[12]; 主要从字符串哈希、protobuf 序列化复合特征类型、值压缩等方法优化内存占用, 缓解 Redis 扩展成本高的问题. Ralf 将模型的准确性作为评估特征质量的重要指标[13-14], 提供了特征消费新鲜度和消费成本间的最佳权衡. 本文研究聚焦于大规模在线特征的版本管理和消费, 设计并开发了基于内存的在线特征存储. 本文的主要贡献有:

(1) 基于上述总结的系统设计要点, 开发了基于内存的多版本特征存储引擎 FeaDB, 提供一套特征版本管理原语. 满足特征从生产到消费的在线管理需求, 包括特征的注册发布, 多源特征摄入, 多版本管理与在线消费.

(2) 采用索引技术优化特征检索过程. 提出了加速单版本查询的内嵌版本索引列表的扩展哈希表索引和支持版本集查询的 Crosshint 索引, 确保查询效率受新鲜度差异影响小. 实验证明两者在读延迟方面都有不同程度的优化, 且 Crosshint 相对哈希索引查询性能更稳定.

(3) 提出了基于时间戳的特征回收方法. 缓解了由大量旧版本特征挤压内存空间及降低版本查询效率的问题.

1　在线特征存储 FeaDB

1.1　系统架构

FeaDB 系统架构见图 1. 特征在接入前需要被注册发布到特征存储中; 其上游系统接入批、流处理系统, 数据流或批量历史数据经预处理、计算转换为特征, 被加载到存储中; 在模型推理阶段, 当有事件请求时, 模型服务系统检索当前时间约束的实时版本及其他特征用于预测任务, 同时可能有多个服务系统接入消费模块. FeaDB 包括特征注册、存储模块、特征访问 (消费) 层、回收数据块的后台线程和分配器. 工程师编写的自定义特征, 如果没有共享和管理, 会导致重复工作和缺乏定义的一致性, 难以应对规模化特征管道并维护特征的可组合性和可扩展性. 特征存储允许特征在特征注册模块发布特征——用户在特征库中编写特征定义元数据, 包括特征转换、版本、所有者等; 特征注册模块间接跨用户和服务提供了共享渠道. 预计算的转换结果被写入存储, 向下游模型提供服务以满足不同的查询延迟要求, 需要一套版本管理原语以向消费层提供版本检索功能. 存储内部由版本集、索引、用于块回收和分配的空间管理器组成. 特征消费层向下游推送最新版本特征.

1.2　数据模型

1.2.1　特　征

特征演进是由数据集更迭以及模型更新引起的, 特征版本为数据源在某一时刻从上游生成/观察到的带有时间戳的特征. 采用时间序列数据表征特征动态变化, 特征的最小更迭单位——特征版本, 可表示为 [Uid, Time, Value]. Uid 唯一标识一个特征记录; Time 表示时间戳, (时序) 特征时间或系统分配的时间戳字段标识一个版本; Value 为该特征版本的特征数据. 特征版本的时间字段在时间点连接 (point-in-time-join) 中使用, 以确保最新的特征值被推送到模型.

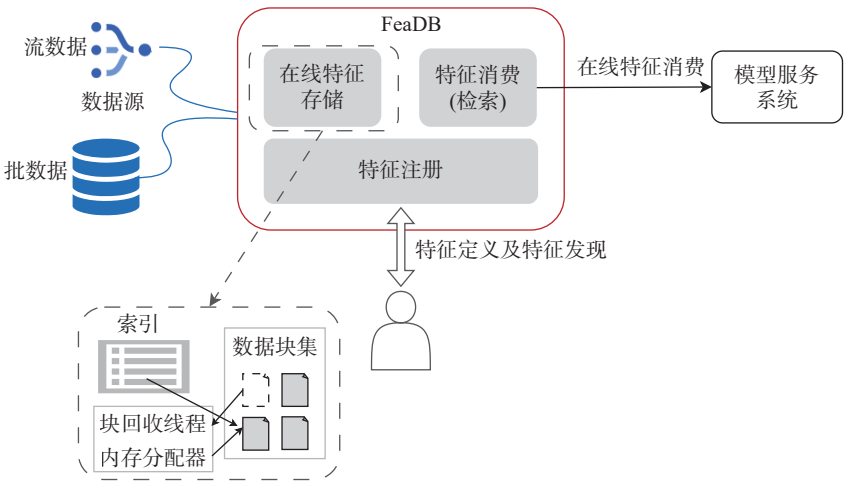

图 1 FeaDB 设计架构图

Fig. 1 FeaDB architecture

1.2.2 特征集与特征快照

用户也可针对模型推理任务创建一组特征集, 在特征集上建立版本快照. 特征集 (feature set) 是一组特征的只读集合, 用户可为每个模型版本注册一个或多个特征集, 用于在推理阶段向模型推送满足新鲜度约束的在线特征, 通常以快照为逻辑单位向下游任务推送数据, 其中快照定义为特征集在某时间范围的一组特征值. 特征集在创建时需要指定 Uid 列表、TTL (time-to-live)、存留策略. TTL 表示快照内的特征时间范围, 存留策略为特征集中每个快照的存留期限, 当 TTL 超过存留期限范围时自动删除版本快照. FeaDB 支持特征集创建、快照读/删除操作.

当特征集被创建后, 不再原地更新值或插入特征, 只进行整体快照版本的更迭, 根据设定的 TTL 定期触发一组版本快照的生成. 用户也可显式执行特征集快照删除操作, 只删除快照, 保留特征值, 快照机制的实现见 1.5.1 节. 特征集的概念适用于管理一组模式较为固定的特征, 其中特征可以被多个特征集共享, 当所有包含该特征的特征集都被删除时, 才会删除该特征. 通过特征集快照, FeaDB 能够重现任意特征集在过去某个特定时间段的特征状态. 在线推理阶段通过快照机制生成时间点正确的特征集以避免数据泄露, 确保未来的特征值不会被泄漏给模型.

1.3 原 语

FeaDB 通过提供特征注册、存储、历史检索, 在线消费等来创建自定义特征管理工作流, 实现特征从生产到消费的数据管道. FeaDB 提供特征版本插入, 时间点正确的版本查询 (单版本与范围查询) 以及面向特征集的操作, 包括特征集创建、快照检索、快照删除等. 用户可通过快照查询来检索在线特征, 并推送到模型服务系统. 用户注册特征集作为某个模型版本的数据集, 设置快照创建的 TTL 及存留策略. 通过快照读方法或范围查找方法检索时间点之前的最新快照版本 (子集). FeaDB 不支持对单特征的删除操作, 用户需要建立待删除特征集, 对特征集执行统一删除. 表 1 列出了主要操作.

1.4 存储引擎设计

重新设计存储引擎来满足预期性能目标, 将从索引管理、存储布局、空间回收策略三方面介绍 FeaDB 的主体设计. 这三个重要模块都会对特征的读写性能造成影响, 同时彼此相互制约, 因此涉及的一些技术难点将在 1.4、1.5 节中分别讨论.

表 1　FeaDB 提供的特征管理原语

Tab. 1　Feature management operation in FeaDB

操作	函数	解释
批量插入	InsertFeature ({Uid,value,time})	插入一组特征版本
点查询	LookUp (Uid, Time)	特征的单版本查询
范围查询	RangeLookUp (Uid, [startTime, endTime))	特征版本在[startTime, endTime)范围查询
获取特征最新状态	LatestVersion (Uid, Time = currentTime)	特征在Time前的最新版本查询
创建特征集	CreateFeatureSet ({Uid}, FeatureSetName, TTL, <Retention Policy>)	注册特征集对象, FeatureSet为一组Uid的集合
读取特征集指定时间窗口的历史状态	SnapshotLookup (FeatureSet, TTL)	特征集在TTL范围的只读快照查询.
在线特征推送	GetOnlineFeatures (FeatureSet, Time, {Uid})	从快照版本中选取子集
删除历史状态	DeleteSnapshot (FeatureSet, TTL)	删除特征集在time版本的快照

1.4.1　优化版本查询的索引设计

在信息检索领域中, 加速检索过程的技术手段有索引、缓存等. 考虑到为快速适应不同任务间特征检索的新鲜度差异, 传统基于频率的缓存替换算法不符合要求, 需要提出量化版本分布的指标并设计精确的自适应替换策略即时响应分布变化, 以提高命中率和查询性能稳定性, 除了淘汰策略, 缓存大小等变量也会影响缓存命中率, 同时缓存无法解决冷启动问题. 综合上述考虑, 本文选择采用索引来加速版本查询过程.

索引设计要求: ① 支持版本点查与范围查询. ② 空间利用率高. 批量更新操作和旧版本索引条目回收对索引结构影响小. 即减少频繁的索引更新及其带来的昂贵的分裂、合并操作. ③ 并发度高, 读写阻塞率低. ④ 查询延迟与新鲜度无关. 基于 4 点要求, 最终选择空间利用率和并发友好的动态可扩展哈希作为基本索引结构, 相比树类索引和静态哈希索引, 其节点 (桶) 分裂 (重哈希) 代价更低. 然而版本粒度的索引构建会引入频繁的桶分裂代价, 且重哈希过程会锁住整个索引导致并发读降低; 再者, 可扩展哈希本质上可理解为物理聚集的版本链, 新鲜度差异影响查询效率. 在此基础上提出索引优化方法, 索引页粒度而非版本粒度来减少桶分裂频率, 查询路径被分散到不同桶中增加并发度; 同时, 将版本链取代为页粒度的版本索引列表, 大大缩减版本搜索路径, 提升查询稳定性.

考虑到每次对只读特征集快照的版本查询都要在哈希索引中重复同样的检索过程, 查询效率低. 提出了特征集快照索引 Crosshint, 即时返回快照版本, 由于索引对象只读, 也不会改变 Crosshint 结构. 下面详述两种索引原理, 在 2.1 节中阐述采用两种索引的特征检索过程.

动态可扩展哈希表是一种动态哈希技术[15]. 相比于静态哈希表, 动态哈希表随着插入 key 数量的增加支持动态的桶增长 (分裂)、桶聚合操作; 空间灵活且内存利用率更高. 可扩展哈希表由以下几部份组成.

(1) 哈希目录: 为一个指向数据块 (桶) 的指针数组. 哈希目录倍增扩张, 因此, 其长度总是 2 的幂.

(2) 哈希桶: 存储哈希键值. 目录指向桶, 当局部深度小于全局深度时, 一个桶可能由多个指针指向它. 图 2 右侧为示例的 4 个桶.

(3) 全局深度 (global depth): 跟目录相关联, 表示通过哈希值的前“全局深度”个比特位作为哈希目录的索引. 图 2 中哈希目录条目为 8, 全局深度为 3.

(4) 局部深度 (local depth): 每个桶关联一个局部深度值, 表示该桶内所有的哈希键的前“局部深度”个比特位是相同的. 图 2 的每个目录索引的标红的比特数表示其局部深度, 若局部深度小于全局

深度, 表示可以有多个条目共享同一个桶.

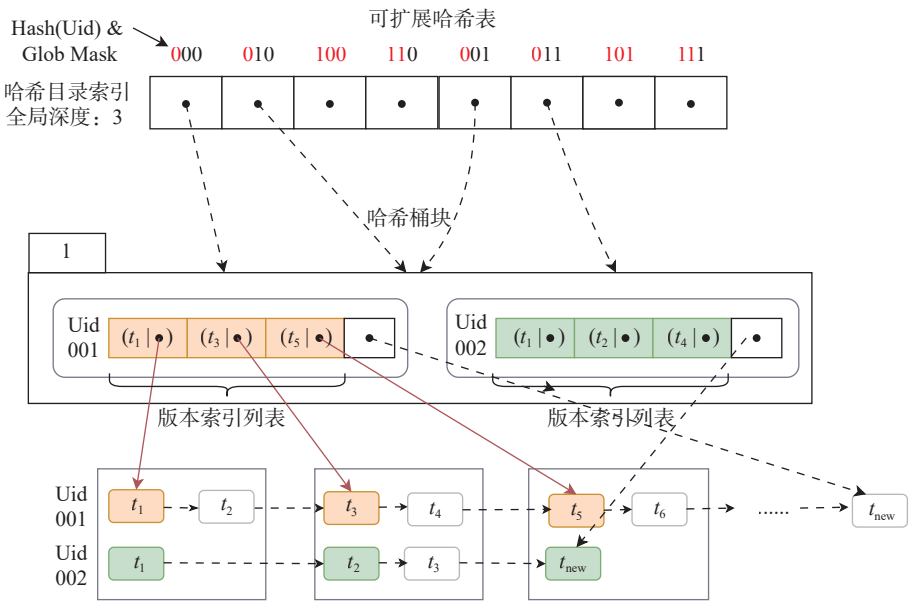

图 2 内嵌版本索引列表的动态可扩展哈希表

Fig. 2 Extendible Hashtable with embedded version index array

当桶溢出时, 表示现有的局部深度不足以区分“桶大小”的哈希键, 因此, 局部深度递增, 桶分裂, 桶内的数据被重新分配到两个桶中. 桶分裂后, 当该桶的局部深度大于全局深度时, 全局深度递增, 且哈希目录倍增. 在哈希查找中, 在哈希目录中通过哈希键查找哈希桶, 匹配桶内数据.

FeaDB 中可扩展哈希表的索引值为块粒度的版本索引列表, 即若该特征的某个版本在某块中出现, 则将首次插入的特征值地址插入哈希表. 在哈希表中, 每个 Uid 的桶槽 (即桶的每个条目) 记录该特征在块中的首个版本地址. 图 2 显示了 Uid 为 001 的特征版本链, 其在 3 个数据块中首次出现的版本分别为 t_1, t_3, t_5. Uid 的哈希桶条目中记录这 3 个版本及其元组地址. 当查找该 Uid 的某个版本时, 通过哈希索引取得其版本索引列表, 在列表上进行二分搜索定位所在块的首个版本地址, 通过元组的 next 字段遍历版本链直到块尾即可. 该索引可避免过长的版本链遍历 (由指针追踪造成的性能降低), 从而将搜索范围减小到块内的短链搜索. 考虑到 Hash 条目不宜占过大的空间, 每个特征限制其版本索引列表长度为 8. 当查询版本未落入该特征“索引窗口”时, 选择替换掉最旧页的索引元组; 同时, 块中嵌入的轻量 min-max 索引也能够实现结果集剪枝, 减少无效的块遍历, 具体见 1.4.2 节.

Crosshint 索引基于特征集快照检索设计. FeaDB 用两个哈希表分别存储特征集的 Uid 信息和 Crosshint 索引, 见图 3. Crosshint 索引特征集在某个版本的快照数据. 具体实现中, Crosshint 的索引键为“特征集名, 版本”, 如图 3 中“F_1, t_1”的哈希值为特征集 F_1 在版本 t_1 上的 Crosshint 索引. Crosshint 版本索引使用一个指针数组实现, 每个指针节点定位到具体的特征版本. 当查询特征集快照时, 根据哈希索引读取 Crosshint 指向的特征集版本. 当快照被移除时, 只删除 Crosshint 索引而无需改动任何数据.

1.4.2 存储布局

特征版本存储结构可采用版本链或顺序组织, 基于对版本回收效率和内存利用率的考量, 采用顺序组织的方式、更新的版本以追加模式写入内存块中, 块写满则分配新块, 以块为单位回收旧版本. 采用这种方式的优点是版本回收效率高且几乎无内存碎片, 减少指针追踪 (pointer chasing) 隐患. 弊端

是版本分散到不同块中, 需要依赖索引设计优化版本读, 以缓解版本随机读带来的查询延迟损耗, 见 1.4.1 节的索引设计. 图 4.1 展示了包含数据块和索引块在内的存储布局.

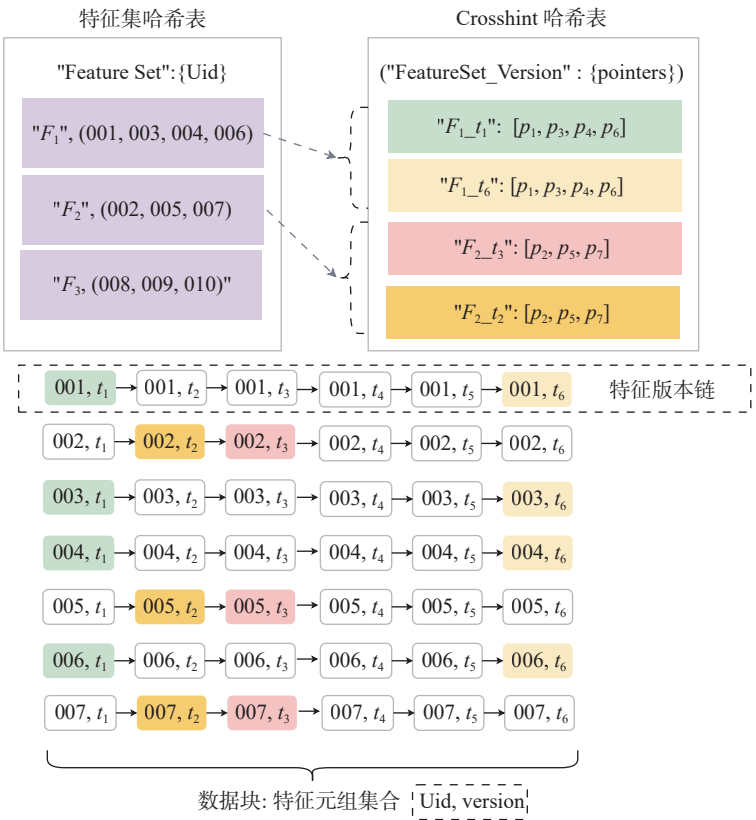

图 3　特征集和 Crosshint 哈希索引

Fig. 3　FeatureSet and Crosshint Hashtable index

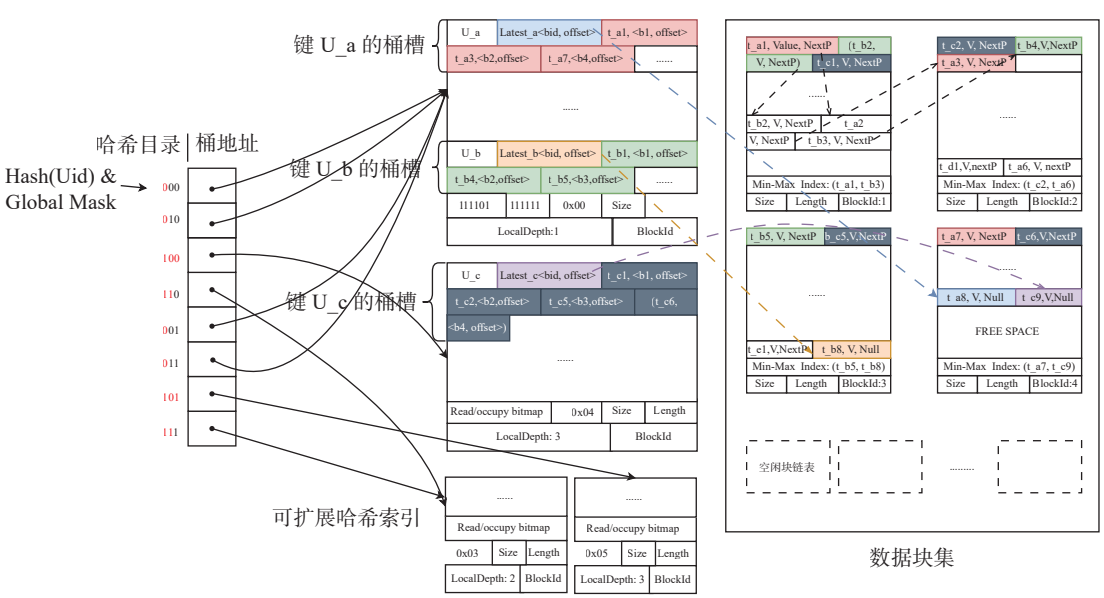

图 4　FeaDB 内存存储布局

Fig. 4　FeaDB in-memory storage layout

特征数据以 4 kB 大小内存块为单位进行管理, 由块内元组数据和其他元数据构成. 其中每个元组结构分为数据长度、元组、下一个元组版本地址 (next 字段) 3 部分. 通过 4 个字段 (Uid, time, value, refCount) 表示一个元组结构, 其中 refCount 字段表示该特征所属特征集数量, 当特征集删除时该字段递减, 当字段减为 0 时 value 字段更新为删除标记. 当 value 为变长字符且其长度超过块大小的 1/3 时, 为该元组 value 单独分配一个块, 采用间接方式访问, 此时 value 为指向该块的指针变量. 下一个版本地址由 (块号, 块内偏移量) 组成, 通过该字段形成 Uid 的一个版本链.

块内元信息包含块号、块内有效负载、块内元组数量及 min-max 索引. min-max 索引为块内第一个元组和最后被插入元组的时间戳 (默认数据被有序插入, 该字段也可作为块创建和块满时间). min-max 索引的好处有: ① max 值可标记为该块时间戳, 作为块回收基准; ② 块粒度索引列表的索引范围有限 (8 块), 当查询范围超出其索引范围时, 可通过 min-max 索引快速过滤结果块集.

存储索引的数据块可分为哈希目录块和桶块两部分. 哈希目录为指针条目数组, 每个条目索引到相应的桶. 当现有全局深度不能区分 Uid 时, 桶分裂的同时目录以倍数扩增. 其哈希目录结构如图 4 所示. 块内字段包含: 块号, 全局深度, 哈希值的前“全局深度”比特位索引哈希键; 桶数量和每个桶的局部深度; 哈希目录大小和索引到桶的指针数组.

每个桶存储哈希键值: 哈希键为 Uid, 值为相应特征链中在各个块中首次出现的版本地址以及最新版本地址. 每个键值占据一个槽, 在一个 4 kB 的索引块中为每个桶分配 32 个槽, 桶满时触发一次分裂操作, 桶中索引将被重新分配. 桶内的每个槽包含键值信息: Uid, 最新版本地址和块首个版本地址列表 (版本索引列表). 桶内元信息除了块号、有效槽负载、已分配槽长度等块信息外, 其他定义的字段包含该桶的局部深度、桶中哈希值 (该字段标识索引到该桶的哈希值)、可读性位图 (该 Uid 是否有效)、占用位图 (该 Uid 是否被删除) 等.

1.5 内存空间管理

1.5.1 空间约束的内存管理机制

索引溢出或数据块剩余空间不足时, 块管理器向分配器申请 4 kB 大小的块, 分配器取出空闲链表头的空闲块; 当块被回收时, 分配器将其插回空闲块中. 分配器为空闲链表的大小设置上下限, 当小于下限时分配器向内存申请空间, 大于上限时将多余的块释放交给操作系统管理. 这样可避免剩余空闲块过少影响块分配效率, 同时减少因空闲块过多造成的空间利用率低的问题.

1.5.2 基于时间戳的版本回收策略

特征集的版本回收包含 3 种方式: ① 移除某个版本的快照只需删除对应的 Crosshint 索引. ② 当特征集版本的数据用户在创建特征集时设置存留策略 (即特征集快照有效期) , 版本回收线程将定期扫描特征集中是否包含过期快照, 并根据用户设置进行版本数据删除或持久化. 对过期特征版本值更新为删除标记, 真正的删除延迟在块回收步骤实施. ③ 用户显式调用特征集删除, 该特征集上所有数据及其快照都会被删除.

块回收. 由于旧版本的访问频率更低, 可将不再推送的过期特征删除, 避免挤压内存空间. 释放旧版本引发块空间回收时, 需要尽可能少的修改元数据, 且数据块的回收也会引入索引结构可能的合并代价. 后台回收线程定期扫描, 若旧块时间戳 (min-max 索引的 max 字段标记) 超过存留时间期限, 则将该块回收, 由分配器清空该块并插入空闲链表中. 由于旧块中的所有数据都可看作是“过期”的 (由 TTL 标记和存留期限共同决定), 因此基于时间戳的块回收机制可确保是安全的, 即回收查询操作不存在交集. 块回收的优势除了增加内存利用率, 实验证明还可以增加查询效率, 减小消费延迟, 见 3.3 节性能评估部分.

2　操　　作

2.1　在线特征检索

2.1.1　单特征历史版本检索

查询步骤: ① 哈希索引查找 Uid, 得到版本索引列表 (即该 Uid 每个版本所在页的首个记录地址 (块号, 块内偏移)), 对列表进行二分查找, 得到满足检索条件的页中首次出现的版本地址. ② 对于索引得到的每个结果由 next 字段进行页内的链表遍历, 查找符合检索条件的元组; 由于哈希索引的页面范围是有限的, 若该特征索引的最后一页符合检索条件, 则后续页还需要进行页遍历, 通过 min-max 索引判断该块内是否有符合检索条件的结果集.

2.1.2　时间点正确的特征集快照检索

查找 Crosshint 在 (特征集, 版本) 的快照. 其中快照建立的过程满足 2.1.1 节中的过程. 查找 Crosshint 哈希时, 若已创建符合时间范围内的快照, 则返回 Crosshint 指向的版本值; 若未找到对应的快照, 则搜索时先查询已有快照中属于时间戳上下限的快照版本 Crosshint(f_s, t_j), 其中 $t_i \leqslant \mathrm{TTL} \leqslant t_j$, 遍历 t_i 到 t_j 之间的版本链定位到所属时间范围的特征集. 通过该方法可确保查找到最近的时间范围不超过检索条件的版本集.

2.2　特征插入

特征插入大致分为写入、插入版本链、更新索引 3 个部分. 插入步骤为:

(1) 根据块管理器得到上次插入结束的位置, 写入新元组并更新块管理器维护的写入位置变量.

(2) 若该块剩余空间不够, 则更新满块的 min-max 索引; 块管理器向分配器申请新块, 写入新元组并更新该块的 min-max 索引.

(3) 和上个版本的元组建立链接. 哈希索引查找 Uid, 根据最新版本索引得到上一个版本地址 (块号, 块内偏移量), 更新上个版本元组的 next 指针为新元组地址.

(4) 更新索引. 若新元组地址的块和上个版本的元组所在块不同, 则更新版本索引列表, 作为新块首个出现的版本; 同时, 更新索引对应条目的最新版本地址字段. 详细的特征版本插入流程见图 5.

2.3　删　　除

2.3.1　删除快照

删除快照时只需移除 Crosshint 哈希索引的对应条目, 其他结构不做变化.

2.3.2　删除特征集

FeaDB 只支持以特征集为粒度的删除. 若需要删除单个特征, 则需要建立包含单个特征的临时特征集, 再执行特征集删除操作. 在删除特征集时, 首先移除相关的 Crosshint 索引以删除在该特征集的特征快照. 对特征集中相关特征的引用数减 1, 当引用数为 0 时对该特征 value 字段标记为删除, 待块时间戳超过存留时间时该块中所有数据被回收.

3　实验评估

3.1　实验环境

实验在 Macos Ventura 13.0.1 操作系统上进行. 处理器型号为 Intel Core i5 @2 GHz 4 核, 内存为 16 GB.

3.2　单特征消费性能评估

实验特征数据选取时间范围为 2023 年 5 月 1 日 9：00—5 月 3 日 9：00 (共 48 h), 10 000 条特征,

其中每个特征的版本数在 100 ~ 1 000 之间, 使用 32 byte 整形类型作为特征值, 平均每个特征在此时间范围内生成大约 300 个版本. 总数据量约 44 MB. 在 3 组负载下分别评估读线程数的单个操作的读延迟, 实验结果见图 6.

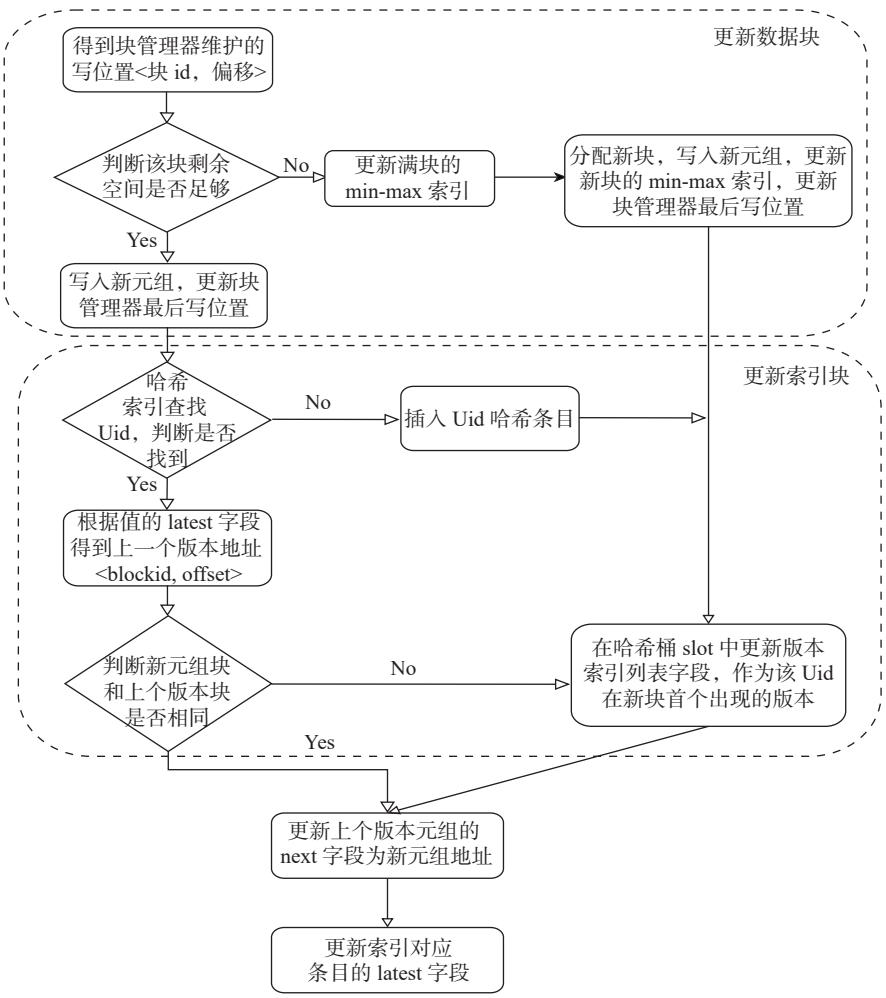

图 5 特征版本插入流程

Fig. 5 Feature version insertion procedure

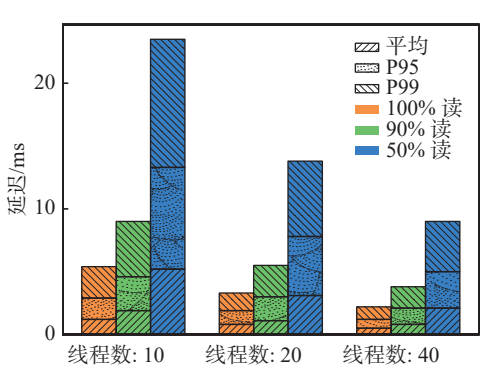

图 6 不同负载下版本查询延迟

Fig. 6 Read latency per lookup with different payloads

结果符合推测, 写会影响读性能. 即使并发度增加会降低单个操作的延迟, 但随着写负载量增加, 并发带来的优势降低. 读线程数量越多, 写对读的影响越大. 原因是写特征版本要更新索引的最新版本地址和新块 Uid 出现的首个版本地址, 因此会出现锁竞争, 当读线程增多时, 竞争的索引桶数增加, 进一步影响读延迟.

3.3　批量特征窗口消费性能评估

系统内部实现范围查询采取两种途径: 基于可扩展哈希的时间范围查询以及在预定义的特征集上基于 Crosshint 的快照版本读. 设置特征集将特征分组, 所有查询特征键 (Uid) 均属于某批特征集数据, 设置 TTL 为 2 h, 存留策略为前 12 h, 则每个快照版本内数据时间范围为 2 h, 同时只保留前 12 h 的数据. 比较特征在 48 h 内的使用范围查询和快照查询间的性能差异. 以 30 min 为窗口, 对随机的批量特征做 100 次范围查询后进行一次延迟统计, 实验结果见图 7.

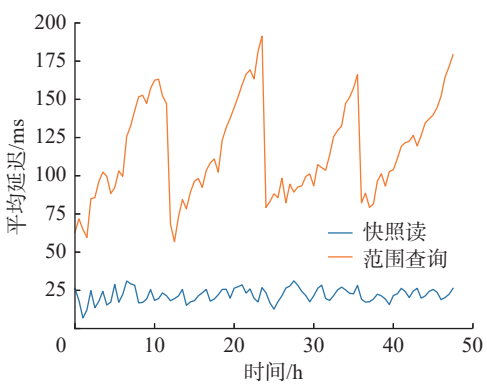

图 7　哈希范围查询与快照读的性能差异

Fig. 7　Hash range lookup performance vs. snapshot read performance

图 7 结果表明, Crosshint 索引的读显著优于哈希表索引读, 快照读性能更稳定, 不受查询时间戳影响, 而基于哈希的范围查询受时间影响较大. 这是由于随着查询时间由旧到新 (在实际情况中, 等价于写入数据量的增加), 当查询的特征版本所在块超过哈希表中版本索引列表的索引范围时, 即超过默认设置的 8 个块时, 需要遍历后续块的 min-max 索引来定位结果集, 增加了读延迟. 观察到图 7 哈希索引查询延迟的间断性骤减, 是因为阶段性的块回收使得块数量减小, 查询时间窗口退回到索引范围内, 避免了块间遍历, Crosshint 也有相应的性能提升, 但显著性不如哈希查询. 间接性表明了旧版本特征回收对查询性能提升的重要性.

4　结　　论

机器学习模型性能很大程度依赖于其特征质量; 同时, 在 AI 辅助的在线决策应用中, 决策结果需要在线特征即时输入以反映真实世界数据分布的变化. 因此, 在数据管理层面, 对特征的实时摄取和消费提出了更高的要求. 为响应这一挑战, 特征存储需要为特征版本管理和实时更新提供保证, 以协同上游的数据摄取管道, 为模型服务系统提供数据动力. 本篇工作提出了 FeaDB——基于内存的在线特征存储引擎, 核心是为上游模型推断任务提供低延迟的特征消费. 设计了两种优化版本查找的可扩展哈希版本索引和 Crosshint 特征集索引, 实验证明 Crosshint 查找快照可以获得更好且更稳定的读性能; 提出了只读快照机制, 相对于哈希检索能够实现更高效的批量特征推送; 分配器采用基于时间戳旧块回收策略, 提升了块回收效率, 同时减少内存空间消耗.

[参 考 文 献]

[1] ZAHARIA M, CHEN A, DAVIDSON A, et al. Accelerating the machine learning lifecycle with MLflow [J]. IEEE Data Engineering Bulletin, 2018, 41(4): 39-45.

[2] LUO Z, YEUNG S H, ZHANG M, et al. MLCask: Efficient management of component evolution in collaborative data analytics pipelines [C]// 2021 IEEE 37th International Conference on Data Engineering (ICDE). IEEE, 2021: 1655-1666.

[3] SCHLEGEL M, SATTLER K U. Management of machine learning lifecycle artifacts: A survey [J]. ACM SIGMOD Record, 2023, 51(4): 18-35.

[4] GHARIBI G, WALUNJ V, RELLA S, et al. Modelkb: Towards automated management of the modeling lifecycle in deep learning [C]// 2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE). IEEE, 2019: 28-34.

[5] SCULLEY D, HOLT G, GOLOVIN D, et al. Hidden technical debt in machine learning systems [J]. Advances in Neural Information Processing Systems, 2015, 2: 2503-2511.

[6] JEREMY H, MIKE D B. Meet Michelangelo: Uber's Machine Learning Platform [EB/OL]. (2017-09-05)[2023-06-30]. https://www.uber.com/en-TW/blog/michelangelo-machine-learning-platform/.

[7] WILLEM P, MIKE D. Feast: An open source feature store for machine learning [EB/OL]. (2021-01-21)[2023-06-30]. https://feast.dev/blog/what-is-a-feature-store/.

[8] CHEN C, YANG J, LU M, et al. Optimizing in-memory database engine for AI-powered on-line decision augmentation using persistent memory [J]. Proceedings of the VLDB Endowment, 2021, 14(5): 799-812.

[9] ORMENISAN A A, ISMAIL M, HAMMAR K, et al. Horizontally scalable ml pipelines with a feature store [C]// Proceedings of the 2nd SysML Conference. Palo Alto, CA, USA, 2019.

[10] FAGIN R, NIEVERGELT J, PIPPENGER N, et al. Extendible Hashing—a fast access method for dynamic files [J]. ACM Transactions on Database Systems (TODS), 1979, 4(3): 315-344.

[11] NETFLIX. System architectures for personalization and recommendation [EB/OL]. (2013-03-27)[2023-06-30]. https://netflixtechblog.com/system-architectures-for-personalization-and-recommendation-e081aa94b5d8/.

[12] ARVAZ K, ZOHAIB H. Building a gigascale ML feature store with redis, binary serialization, string Hashing, and compression [EB/OL]. (2020-11-19)[2023-06-30]. https://doordash.engineering/2020/11/19/building-a-gigascale-ml-feature-store-with-redis/.

[13] SARAH W. Ralf [EB/OL]. (2022-03-13)[2023-06-30]. https://github.com/feature-store/ralf/.

[14] MOHANTY P, KRISHNASWAMY S, CHOI E. Automated Cache Hierarchy for Feature Stores [R]. CA: University of California, Berkeley, 2021.

[15] ORR L, SANYAL A, LING X, et al. Managing ML pipelines: Feature stores and the coming wave of embedding ecosystems [EB/OL]. (2021-08-11)[2023-06-30]. https://arxiv.org/pdf/2108.05053.pdf.

(责任编辑: 李万会)